Sam Ainsworth

Post-process anti-aliasing of video data

Computer Science Tripos

Churchill College

April 8, 2014

Proforma

Name:	**Sam Ainsworth**
College:	**Churchill College**
Project Title:	**Post-process anti-aliasing of video data**
Examination:	**Computer Science Tripos, May 2014**
Word Count:	**11637**[1]
Project Originator:	Sam Ainsworth
Supervisor:	Dr Thomas Perry

Original Aims of the Project

To create a device to ameliorate aliasing artefacts for realtime rendered video by processing based only the output image stream rendered by a system. This will be done by using an FPGA-based processing system to perform anti-aliasing as a post-process by blending based on particular image features found in each output image using a technique called Morphological Anti-Aliasing. An image source will be provided by a video input, which will then be processed and output from the device to a television or monitor.

Work Completed

A device has been designed and implemented on an Altera development board to take in video input over a composite connection, process it using the Morphological Anti-Aliasing algorithm to remove aliasing artefacts, and output this over VGA to a monitor. The processing was done on twelve soft-processors on

[1]This word count was computed using texWordCount.pl -c dissert.tex https://www.comp.nus.edu.sg/ kanmy/software/texWordCount.pl

an FPGA, along with custom hardware to form a system-on-programmable-chip, to achieve processing at a rate of 8–15 frames per second.

Special Difficulties

None.

Declaration

I, Sam Ainsworth of Churchill College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date

Contents

List of Figures

Chapter 1

Introduction

Rendered 3D graphics can suffer from artefacts as a result of only sampling once per pixel, known as aliasing, as seen in Figure 1.1. Conventional solutions to this employ sampling multiple times in areas where many polygons cover the same pixel. However, imagine the common case where we cannot do this: often it is too expensive to do so, or we do not have the hardware support required. This project considers the case when we do not have access to the hardware at all, for example with embedded systems. When the only information we have access to is the video output of the device, we must use other means to smooth the edges of images.

The practical use behind this project is with video games consoles rendering real-time 3D images. Many lack the power or dedicated hardware for anti-aliasing on-system, so a device between the output of that system and the television that improves image quality is of great use. The goal of this project is therefore to produce a device to output a video feed of improved quality with respect to image artefacting, at as reasonable speed as is possible, though not necessarily the speed of the original input.

To ameliorate aliasing artefacts with only the video output available, I used an Altera development board to design a system to perform morphological anti-

Figure 1.1: A bitmap line, before and after morphological anti-aliasing. The latter line is visibly smoother despite being at the same resolution.

aliasing (MLAA) [11], a high quality, readily parallelisable post-process video technique, on a video input, and then output it again over a VGA port. This was performed using a combination of hardware modules running on the development board's FPGA, including twelve Altera Nios II soft-core processors with custom instructions to improve performance of the software running on them, to perform the anti-aliasing.

1.1 Morphological anti-aliasing

An algorithm for smoothing edge artefacts based on the properties of an image is morphological anti-aliasing (MLAA), as first described in a paper by Reshenov of Intel [11]. The algorithm works by finding discontinuities in colour, trying to find L shapes in these discontinuities, fitting lines across the L shapes, and smoothing based on the result.

In more detail, the following stages are performed:

1. Find horizontal discontinuities between pixels, moving from top to bottom along each vertical line (considering the edges of images to be continuous).

Figure 1.2: Horizontal discontinuities

2. Find vertical discontinuities between pixels, moving from left to right along each horizontal line.

Figure 1.3: Vertical discontinuities

3. Find horizontal lines in the discontinuities. At each end, follow which side (up/down) has an edge (if one exists) by half a pixel[1]. This leaves us with either a U shape, an L shape, a Z shape or an I shape, depending on the direction of each end.

Figure 1.4: A U and an L shape fitted to the horizontal discontinuities.

4. Ignore the I cases (there's no new line to fit), and split Zs and Us into two Ls. Fit straight lines across the far edges of each L shape. Set the new colour of each pixel to be the area average of the colours each side of the line.

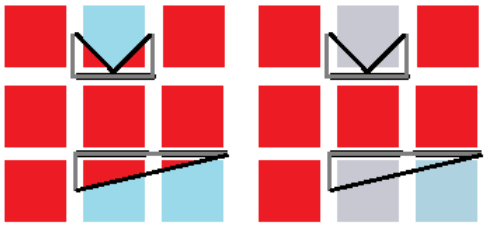

Figure 1.5: Horizontal processing. The pixels with lines through take on a slightly more reddish colour, based on the area the lines go through.

5. Do the same for vertical discontinuities.[2]

Figure 1.1 shows an example of the algorithm being used on a diagonal line.

[1]We only go up by half a pixel because we only wish to work at the pixel level, and only wish to fit to lines that can't be accurately resolved by sampling at pixel-level detail. If we went up more than that, we'd be adding structure to the image that was clearly not intended in the original scene.

[2]Pixels can be processed one to four times depending on the number of discontinuities surrounding them. This can result in slight blurring, but by reducing discontinuities in colour tends to produce a more pleasing image.

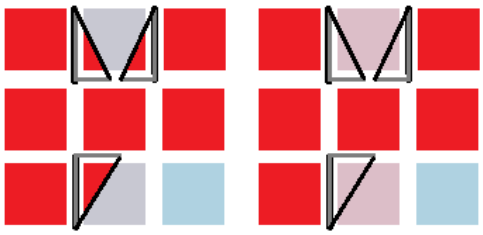

Figure 1.6: Vertical discontinuity processing. Note that the colours the pixels start with are those based on the already processed horizontal discontinuities.

1.2 Prior art

The definitive implementation of the above algorithm is based on the work of Reshetov [4]. Source code is available but it is heavily optimised for Intel instruction sets, using SSE instructions, for example, and is thus difficult to work with. This leaves us with the original paper [11] as the easiest source of information to work with. Sony's Advanced Technology Group have similarly implemented MLAA on the Cell processor of the PlayStation 3, where it has been used in numerous titles [7]. Still, both these implementations are used as an end-process of the rendering pipeline of the specific machine that is doing the rendering, rather than a separate process that can be applied to any video input.

The only existing solution I have found for filtering arbitrary video and outputting it again involves using a capture card for input coupled with a full Desktop PC and powerful graphics card to do the processing and output, via the capturing software PtBi [12]. This works well but is clearly an expensive and very high power solution: using as much energy for anti-aliasing an image as generating it seems excessive, and purchasing capture equipment along with a powerful computer is a high barrier for entry. Such a setup is also likely to be too bulky to fit between a typical console–TV setup.

Chapter 2

Preparation

2.1 Requirements Analysis

The device must:

- Be able to accept video from a video-in source, process it, then output it again on a video-out connection.
- Run the MLAA algorithm on the video, at as low latency and high speed as is practical to achieve, to improve image quality[1].
- Be implemented on a prototype board that could lead to a low power and small commercial implementation, such that it would be practical to use between a games console and a television.

The device need not:

- Run at high definition resolutions - this would be difficult to achieve in terms of performance, and hardware with such inputs and outputs would be difficult to procure.
- Output at the same frame rate as the input, though it is desirable to achieve as high frame rates as possible to improve usability of the device.
- Be portable between lots of different development boards.

[1]Video streams tend to show between 25-30 unique frames per second, but useful output can be achieved well below this.

2.2 Parallelism of the Morphological Anti-Aliasing Algorithm

To consider the best method of implementation for such a device, we must consider the performance characteristics of the morphological anti-aliasing algorithm described above; in particular the gains available through parallelism. We should therefore consider the parallelism of each part of the algorithm in turn.

The first two sections as discussed in 1.1, finding horizontal and vertical discontinuities, read from one block of data (the framebuffer), and write to an entirely separate data structure (implementation specific, but logically something that records whether there is a discontinuity between each pixel). If we assume the latter data structure is an array, then each comparison between adjacent pixels doesn't conflict with any other: theoretically, we could run every single check at once on its own CPU.

There is less available parallelism in the sections where we actually process discontinuities, but it is still abundant. Processing horizontal and vertical discontinuities clearly are conflicting operations (a single pixel can have both surrounding it, both of which can alter the value of the pixel), so proper mutual exclusion must exist between these stages. However, within each stage, a discontinuity line can change either pixels directly above or below it, and calculations for the colour of those pixels in turn can be based on the pixels directly above or below the line. Thus, we can't run discontinuity processing on two adjacent lines at once, but any other two lines can be processed in parallel. Thus, we can calculate half of the lines at once, then the other half, and we can parallelise by a factor of $\frac{N}{2}$, where N is the number of lines in that pass. Clearly for reasonable resolutions this is much higher than the amount of processors we could realistically use to process an image, so there should be no issue with efficient parallelisation, at least in theory.

The above means that we should be able to parallelise adequately for any reasonable number of processors. If this were not the case, another potential method of parallelising the MLAA algorithm would be to pipeline by frames: have each processor work on one frame at once, but work on multiple frames concurrently on different processors. This would be a way of increasing throughput, but is inferior to intra-frame parallelisation in that it introduces much higher latency: we still have a full frame's worth of computation on one CPU before we get a frame through. The reason this is a severe issue is that the ideal use case for this project, realtime video gaming, requires low latency to be usable, and thus minimising latency is as important as maximising throughput. This suggests,

then, that this is not a good method to explore.

2.3 Specification

To fit the above requirements, my device would need to be implemented on a development board with both video in and out, and the ability to actually run a form of the morphological anti-aliasing algorithm. The logical choice here therefore was an FPGA development board – the inherent parallelism of the algorithm, as discussed in Section 2.2, could be exploited using a custom hardware setup, and video input and output of video could be achieved without extra hardware. The board I chose to use was the Altera DE2-115 as it:

- Was readily available from the CL as part of the Altera VEEK-MT development board
- Has composite video input and VGA output, along with FPGA blocks available from Altera to drive them.
- Is more powerful than the DE2 I already owned, in terms of the amount of DRAM and logical units. The project needed more DRAM than the DE2 offered (128MB on the DE2-115 vs 8MB on the DE2), for having the stacks and heaps of many processors in memory at once, as well as being able to store enough buffers for deinterlacing and performing the MLAA algorithm off-screen whilst showing the previous frame. Further, lots of logic units were required to instantiate many CPUs, and the DE2-115 has over 100,000 of them – each Nios II CPU is 2000 logic units (excluding cache, and hardware multiply/divide), so there was plenty of space for large hardware blocks.
- Has lots of existing hardware blocks available from Altera to use in projects, and is a commonly used board with relatively good documentation.

It was decided fairly early on in the project, after consulting with both my supervisor, Tom Perry, and Simon Moore, that a pure hardware implementation would be too complex to implement in the amount of time suitable for a project, particularly because the algorithm as described in Section 1.1 iterates over the entire data structure of a frame multiple times in memory. Thus, it was decided to use programmable CPUs to an extent. Since the algorithm is considered embarassingly parallel, that is there is little communication between parallel tasks, it seemed appropriate to exploit this by using a large amount of Altera's Nios II soft-core processors on the FPGA to power the processing of the device.

However, implementing everything in software would be too slow for the implementation to work well, and also wasteful of resources: when it is possible, performance tends to be higher using fixed function hardware [6]. Indeed, since the device needed to do input and output of the video in a useful format to be processed alongside doing the anti-aliasing, not everything could be as easily parallelised in software as the algorithm itself. Altera's library of existing hardware blocks is fairly extensive, including the Nios II soft-cores, RAM controllers, video processing blocks and DMA controllers, and thus it was chosen to use these wherever possible instead of writing custom hardware. The reasons for this are twofold: by using existing hardware blocks it was possible to cut development and testing time down dramatically, and also Altera's implementations are likely to be better optimised in terms of speed and size than my own custom hardware would have been.

Since it was a good match for the algorithm, and due to the low achievable clock speed of FPGAs, it was mandatory to exploit parallelism to achieve good performance for the project. Thus, a firm idea of multiprocessing capabilities was required before the start of the project implementation. The standard Nios II multiprocessing methodology is rather unconventional: separate binaries need to be made for each individual processor rather than uploaded as a single package, making calls between CPUs to support running code on each one. Luckily, Altera provide a tutorial on this system [3].

For parallelism the implications of any caching system must be considered. The Nios II does not support hardware cache coherency between processors, so the size of cache lines and how that would affect concurrency was important for the project. The Nios II supports cache line sizes of up to 32 bytes (eight words), so as long as the same cache line isn't in two processors' caches at once when one writes to that cache line issues are avoidable. If we assume we have 640 horizontal lines of data, then, as long as we work in horizontal lines and flush the cache after finishing this shouldn't cause any issues, as each set of 640 pixels should fit cleanly into cache lines. For cases where this isn't possible (processing vertical lines), to ensure we only access cache lines in one processor between cache flushes a new limit on parallelism is required, which is that we can't process multiple lines which have pixels in the same cache line concurrently.

The Nios II has good compilers for C and C++. Since good performance was likely to require the low level memory management and relatively direct hardware access possible with both of these, C was chosen as the language of choice for writing software for the Nios II MLAA implementation. It was unlikely that the object oriented features of C++ would be useful for the algorithm as described above, so there was no need to consider using C++ even though the option was

available. To make the porting process to the Nios II easier later on, I also decided to write my initial PC implementation of the MLAA algorithm in C.

2.4 Starting Point

I had picked up a good working knowledge of C from a summer internship, but my knowledge of Verilog and System on Programmable Chip design was limited to that in the Part IB ECAD practicals. Clearly designing an entire system from scratch required a very different level of knowledge from being able to follow practicals, so a significant amount of time was required to fully understand how to work the software tools involved.

In terms of existing code, the Morphological Anti-Aliasing algorithm is well documented in Reshetov's original paper [11]. However, while source code is available for Intel's implementation, it is too Intel specific to be of any use in porting to any other system, and thus code for the algorithm needed to be written from scratch.

2.5 Development Environment

Since I was using an Altera development board, Quartus II, Altera's programmable logic device IDE, was the logical choice of development environment, and is available for both Windows and Linux. The DE2-115 also has software available for it to automatically generate Quartus files with the correct output pins and top level module, but this is only available for Windows. It was the latter that made me decide to use Windows 8 as my operating system of choice for the development of this project.

Quartus includes a variety of powerful tools to allow efficient hardware development. QSys, the included System on Programmable Chip design tool, makes interconnect between different modules simple with a shallow learning curve, and adding new modules to QSys is similarly simple. QSys can also automatically assign addresses to hardware, and synthesise arbitration logic for memory mapped hardware. The graphical approach to designing systems also makes designing complex systems fairly quick and painless. Further, lots of Altera IP is already designed to work with Quartus, which is useful when utilising existing modules. A version of Eclipse, the software IDE, is also provided, which is augmented for the design of code for Altera's Nios II soft-processor, and is thus the best option for producing code for the Nios II.

For my initial implementation of morphological anti-aliasing on PC, I used Dev-C++ as I was slightly more experienced with it than Eclipse for C development. However, I then moved to Eclipse for the Nios II, as required by the good integration of tools therein. I also occasionally used Notepad++ for rapid editing of Verilog source files.

2.6 Backups and Version Control

GIT was used over the University Computing Service's MCS for version control and backups, as the MCS is considered a reliable storage location. Clearly only some of the features of GIT were useful for this project: only I was working on the project so the collaboration features were needless, but the version control was used on several occasions to roll back changes, and it also served well as a backup. However, Quartus generates far too many dependent files to have stored the entire project using GIT (over 26000 files, 1GB worth, by the end of the project), so it was only practical to store independent files over the MCS. To make recovery easier, and also in case of missing some independent files amongst the vast quantities of dependents, I also periodically uploaded a folder to Dropbox to make certain I wouldn't lose my project.

2.7 Logging

To keep track of changes to the project, I kept a diary at the root of my GIT repository, including changes, hypotheses on bugs, fixes for bugs, plans for future improvements and performance measurements, all indexed by date. This was useful as a way to keep track of progress relative to the plan document, and record, as reminders, problems at a given time and my thoughts on solutions. It was also useful as a basis for progress emails to my supervisor, however sometimes text wasn't adequate to describe parts of the system at the correct level of detail whilst still keeping the information intuitive, and thus I also used diagrams where necessary to make obvious the design and progress of my system, to keep my supervisor up to date and to eventually use in my dissertation. These were drawn in OpenOffice Draw. The logging system of GIT was also used, but less to keep track of the project and more for logging of changes at each version, to make choosing the correct version to roll back to easier in case of bad changes.

2.8 Development Model

It was infeasible to develop a project of this complexity using the waterfall method with my relative inexperience with System on Programmable Chip design; there would be no hope of being able to test the project as a whole without breaking the project into iterations with various testable features. Thus, I decided to use an iterative development paradigm, whereby I iterated over a basic design, improving and adding new features at each stage, and testing these until I was convinced they were correct. Further, by taking this approach, any extensions, which were of uncertain scope and difficulty at the start of the project, could be added or removed as appropriate from the iteration cycle. Also, it was entirely feasible to have these iterations catered for in the design of the project; many of the proposed features of the design were to improve its speed, so a basic implementation could be produced in significantly less time than the proposed entire system.

The project was therefore broken down into the following sub-tasks, each with a deliverable mini-project at the end:

1. MLAA algorithm running on a PC, to process images. By first writing up a version of the algorithm for PC, I could better understand the algorithm for implementation on my device, and also produce output images from bitmap files to test my implementation, so I could be better assured of correctness when processing video data.

2. Port MLAA to Nios. Getting the above code running on a simple Nios system to get a rough idea of performance, and to work out any issues before adding further complexity.

3. Video In / Out. Finding a method of getting video data in / out of the chip, and thus creating a system that could display video coming in over composite input, and out over VGA.

4. Output of test image with MLAA: test getting video off-processor without worrying about the properties of real video.

5. Connect video in to the MLAA processor. This would allow a complete (though probably slow) implementation of video anti-aliasing and output, which could be tested before any improvements were added.

6. Speed improvements to the above. Any improvements to the above project that didn't involve multiprocessors could be done here, for example code optimisation and use of more custom hardware.

7. Multi-core MLAA. Using multiple processors to improve the speed of the algorithm, and implementing and testing any hardware/software to allow this change in the system.

Chapter 3

Implementation

The following sections discuss the implementation of the project following the plan in Section 2.8, starting with an initial PC algorithm, followed by porting to the FPGA, video in and out, and single and multi-core optimisations, resulting in a fully fledged high speed system.

3.1 MLAA Implementation

A high level description of the morphological anti-aliasing algorithm is given in Section 1.1. The following is a description of the features of my particular implementation.

Whether we have a discontinuity between pixels is decided by comparison of only the three most significant bits of each of the RGB channels (which are 8 bits each in all implementations). If these are unequal then the pixels are deemed to be different colours. This means we avoid processing pixels similar in colour, saving processing time, avoiding blurring fine detail, and achieving a simple discontinuity check: we bitwise `and` each colour value with a mask, and then check if the results are different. If we align our colour data to words we can achieve each check with two `and`s and a comparison, which is a very cheap method of finding such discontinuities.

Clearly the above isn't perfect – bit patterns 11100000 and 11011111 are seen as being discontinuous from each other when arguably they are almost unnoticeably different (224 vs 223), for example, but what we lose in terms of potential slight blurring on some areas of the image, and added execution time from some more edge processing, we gain in not having to perform expensive operations on every single pixel of the image.

We store these values as 8 bit C boolean values in memory in the form of two two-dimensional arrays `horiz_discontinuities[y][x]` and `vert_discontinuities[y][x]`, where `horiz_discontinuities[y][x]` is true iff there is a colour discontinuity between (x,y) and (x,y+1), and similarly `vert_discontinuities[y][x]` is true iff there is a discontinuity between (x,y) and (x+1,y). Testing on the Nios II revealed using 8 bit booleans over 32 bit integers for this purpose was quicker; the cost of sub-word access seemingly being less than the time cost of transferring more data.

To find discontinuity lines, my implementation then moves through these arrays looking for lines of `true` values signifying discontinuities. When these lines are found, we use the other array at the end points of the line to work out which shape of discontinuity we have (two forms of Z and U, four forms of L, and an I), by finding values of `true` either on the same line (pointing upwards perpendicular to the line) or on the line below (pointing downwards perpendicular to the line). If the program finds both, I set the line to be flat for performance – we can imagine choosing based on biggest colour difference, for example, but this would have slowed down computation. Then, taking the imaginary line to start half way up the pixel (for reasons stated in Section 1.1), the area between that line and the edge of each pixel is calculated, so we can calculate the colour the pixel would have had with an area average sample of the "true" line.

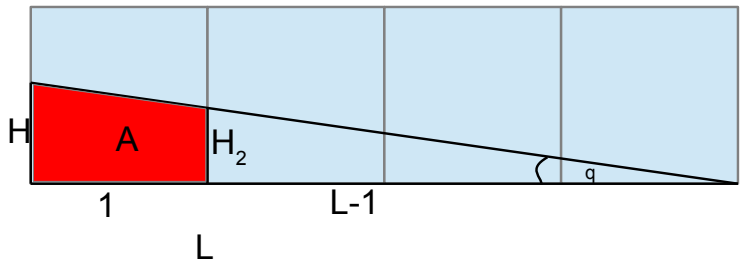

Figure 3.1: Diagram of the area of the pixel we need to fill.

To be able to process line edges in this way, we need to work out the area between the line and the edges of each pixel, and we need to do this efficiently; if we waste computational power the algorithm will be too slow to run in realtime. Thus, we must consider the triangle formed by the line with the edges of each pixel.

Figure 3.1 shows the values we need to consider. A is the area of pixel (x,y)

we must fill with the colour of pixel (x,y+1), which would be the real colour of that part of the pixel assuming the line goes through there. L is the length of the L shape to process (as mentioned in 1.1, we split all other shapes into Ls). H is the height of the triangle – for the start of new lines this will be $\frac{1}{2}$, but the concept generalises to any of the smaller triangles formed along the line, such as that generated by H_2 and $(L-1)$, where H_2 is the distance between the line we are fitting to and the bottom of the pixel at the edge of pixel (x+1,y).

To work out A:
$A = \frac{1}{2}(HL - H_2(L-1))$ by area of right angled triangle.
$\frac{H_2}{H} = \frac{(L-1)}{L}$ by geometrical similarity of triangles.
$H_2 = \frac{H(L-1)}{L} = H - \frac{H}{L}$
And thus, by substitution
$A = \frac{1}{2}(HL - (H - \frac{H}{L})(L-1)) = \frac{1}{2}(H + H - \frac{H}{L})$
$A = H - \frac{H}{2L}$
And we can work out the value of A for the next pixel by setting $H = H_2$.
$A_2 = H_2 - \frac{H_2}{2(L-1)} = H_2 - \frac{H_2 + \frac{H_2 L}{L-1}}{2L} = H_2 - \frac{H}{2L}$
$A_N = H - \frac{(N-1)H}{2L}$

Clearly the above involves performing an initial division of $\frac{H}{L}$ for every line we wish to fit (though we can reuse the value for working out all other values of A_N for that line). This could be concerning, given the relative slowness of division on most hardware, for an implementation working on real video that we wish to process as fast as possible. However, the Nios II does support an optional integer hardware divide unit, and with this enabled the difference between performance using the integer divide and performing an arbitrary operation instead (in this case a right shift) in this step of the algorithm provided negligible performance differences on actual Nios hardware. To make correctness justifications easier my initial implementation was floating point based, but for performance reasons this was eventually switched out with a fixed point version of the code.

3.1.1 PC Implementation – Bitmap Images

Images are stored as arrays in memory (24 bit RGB on the PC bitmap implementation, and early builds on the FPGA, and word aligned 32 bit RGBX in later FPGA builds), as part of a C struct which also includes information on the height and width of the image. Figure 3.2 shows the structure of this. Since the PC implementation is for reading and writing of BMP images, many of these fields are bitmap centric – since bitmaps are word aligned at the end of rows we have a field for the width of the bitmap's lines, a filesize (for checking the integrity of

```
typedef struct image{
    int filesize;
    int width;
    int bmp_array_width;
    int height;
    unsigned char* image_data;
    unsigned char* total_file;
} image_t;
```

Figure 3.2: Structure of images for the MLAA algorithm.

the image), a pointer to the start of the BMP including headers, and a pointer to the direct header. The FPGA implementation only uses a subset of these fields (since it isn't working with bitmap files), but to keep compatibility across code both use the same overall image structure.

After researching various libraries for image reading and writing, I decided the quickest option was to write my own bitmap loader, writer and parser for 24bit bitmaps. I chose to process only bitmaps as the format is much simpler than most other image types: raw RGB data is available without transformation. The bitmap format consists of[1] a header followed by the actual data in RGB24 format (eight bits for each). This data isn't word aligned for each pixel, but is for horizontal lines; this means I calculate in my bitmap parser how long each line is and store it in the image struct. I then compare the size of the file with the size of the header plus the length of each line multiplied by the height of the image, to check that an input image really is a 24-bit bitmap, and also as a good test of correctness for the parser.

Using bitmap images as input and output of the PC implementation yielded significant gains over working on Nios straight away; getting direct output at an early stage allowed me to analyse the workings of the algorithm quickly, fixing many bugs that would have been much harder to spot with only test benches giving raw data, because visual output gave a very clear test of the code from analysis of the effects of the algorithm on a variety of images.

3.2 Nios II Soft-processor

The Nios II is a soft-processor designed by Altera for use on their FPGAs. It comes in three configurations in order of increasing size and power: the Nios II

[1] Parsing information from [8].

e(economy), s(standard) and f(fast). I chose to use the Nios II f for this project, partly because of its increased performance due to added hardware such as branch prediction, but more importantly because it supports data caching, whereas the others do not, thus allowing the spatial locality of moving through data arrays to be exploited.

The Nios II has optional hardware multiply and divide, and since my implementation of the MLAA algorithm uses both of these frequently (multiplication for array indexing and division for working out the area to cover of each pixel as described in 3.1, for example), it was advantageous to use these. The cost is using extra logic units on the FPGA, but because I didn't anticipate being logic unit bound on the device it was a natural choice to make use of them. Clock speed and memory bandwidth were anticipated to be the limiting factors, and by using hardware multiply and divide I could achieve more work at lower clock speeds.

Figure 3.3 shows the structure of the CPU at this point along with the blocks directly attached to it. In my initial implementation I used a single processor for MLAA processing. Since both code and data were stored in SDRAM, there is a connection from the Nios block's instruction master and data master to the SDRAM module slave. This is actually a connection over the Altera Avalon-MM memory mapped network-on-chip interface available in Quartus; by using this, arbitration logic was automatically synthesised, making design-end work simpler. Also on the Nios' data master are the timer and JTAG UART modules. The timer allows us to be periodically interrupted with timing information; this was used in this project to measure how long data loading and each stage of the morphological anti aliasing took, both to target optimisations at the correct section of the code (in accordance with Amdahl's law), and to get performance measurements for evaluation. The JTAG UART was used to both load programs onto the CPU, and get standard out from the device, over a JTAG USB input from my development computer.

3.2.1 MLAA on Nios II – Initial Implementation

Porting the code I had written for the PC implementation of the MLAA algorithm was fairly simple, since I had written the original in C, and a C compiler was available for the Nios II. Any platform differences were easily accommodated for using the C preprocessor; indeed, the PC and Nios II projects shared the same C source file for the MLAA processing throughout, with platform specific changes added or removed depending on which preprocessor values were set. However, passing arbitrary input bitmaps to test it was less trivial, as at that point I had

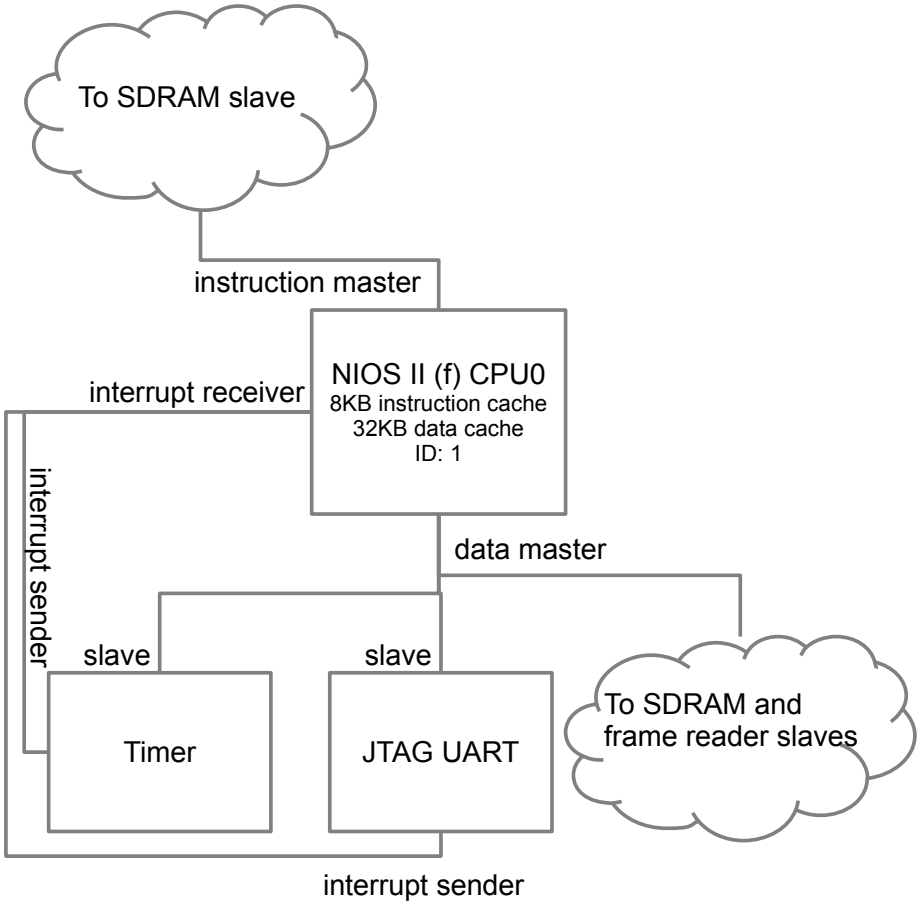

Figure 3.3: Structure of the components initially attached to the CPU.

no way of getting images in and out of the device. Since the image format the algorithm sees is independent of file format (as described in Figure 3.2), and thus bitmaps weren't required to test the device, I generated an image of a red circle in the centre of a black background as a test image as part of the program. Since this would mean I couldn't directly get visual output, I used stdout over JTAG to output the coordinates and values of altered edges to test the algorithm was working correctly. This gave me a good guarantee the code was working correctly without actually seeing any visual data; I avoided trying to connect up video systems at this point, to test the processing isolated from any video faults.

3.3 Video Out

To make use of this device, a method of outputting the anti-aliased video from the CPU was needed. The following section describes the hardware blocks and techniques used to achieve this.

3.3.1 Getting Video in and out of the DE2-115

Before sending video data through the CPU, I wanted to test video input, decoding and output. I found an implementation by a member of the Altera forums [1], which used Altera's Video and Image Processing Suite (VIP) of video processing blocks to take in composite video decoded by the TV decoder chip on the board and output it again over VGA. This was far from being usable to send data in and out of a CPU; there was no data buffering, data could only flow down the specified Avalon-ST data streaming paths, rather than into a CPU or RAM over Avalon-MM memory mapped interfaces, and the data format size was incompatible with 32 bit word based CPUs, but it was a good starting point, and integrated well with QSys.

3.3.2 Altera Avalon-ST and the VIP Protocol

It would have taken too long to have implemented every video module from scratch, and would also have been inefficient: Altera have various video processing modules available for integration with projects using QSys. Thus, I tried to use Altera hardware modules wherever possible rather than coding my own. However, this approach had tradeoffs; Altera's devices use their own formats which must be obeyed when using such modules. The Altera Avalon-ST protocol is Altera's data streaming interface (as opposed to Avalon-MM, the memory mapped interface for working with RAM, for example), which supports unidirectional flow of data,

with packets and flow control. The VIP protocol is the data format for Altera's Video and Image Processing Suite of hardware blocks, including video and control information.

3.3.3 Frame Reader

Altera provide a Frame Reader block for reading out video data from a run-time adjustable address of memory. This reads the data using Direct Memory Access (DMA), having its own data master for the SDRAM, which is much faster than outputting from the CPU via a FIFO, for example. This module can then output data to the video-out module using the Avalon-ST protocol: my particular implementation outputs video using three parallel 8-bit channels for red, green and blue. The data format the Frame Reader requests in memory, however, is RGBX32: 8 bits of red, green, blue, and nothing, which is good for CPU data because it is word aligned: though clearly it is somewhat inefficient in memory usage, wasting $\frac{1}{4}$ of the memory it allocates, by being aligned to 32bit words it is efficient to access. However, this format is incompatible with Altera VIP, which meant that I could not directly input data from an Altera block into memory, and some formatting was required on input data before being placed into memory; this was initially done in software but was later moved to hardware (see Section 3.6).

3.3.4 Clocked Video Out

Altera provide a module for connecting with the VGA output, that deals with clock domain conversion (VGA at $640 \times 480 \times 60$ clocks at 25MHz, whereas much of the rest of my project's hardware runs at 80MHz) and video output. This module is generic for all video out types Altera VIP supports, with no obvious settings for VGA output. Thus, I used the settings from the example project I discussed in Section 3.3.1.

3.4 Video In

The video out system described in Section 3.3 allowed moving video data off of the CPU fairly simply. However, at this stage, I had nothing to show on screen. The following sections move from an initial implementation using a test video generator, to pulling in video over a FIFO and performing MLAA on it, to high speed DMA transfer of video to memory.

3.4.1 Initial Implementation

Test Image Generator

To avoid complicating the process of getting data into the CPU, I initially used a test image generator, an Altera block from the VIP suite, which outputs a stock image over Avalon-ST. This had the useful property of always being the same image, thus being a constant to work against for testing. Real image data, being output from a video device at 60 frames per second, would be lost if we didn't get it into the CPU fast enough. Since the Test Image Generator was flow controlled, I could be sure of not losing data regardless of throughput.

FIFO

To be able to process the test image data on the CPU, a method of moving data from input to a memory location on the CPU was required. The simplest way to do this was by using an Altera FIFO hardware module: the 24 bit RGB video data was loaded into a FIFO 8 bits at a time, from where the CPU could load the data using programmed IO.

3.4.2 Video In Subsystem

The initial, fully functional design of the hardware for video input is shown in Figure 3.4. To get the data into a reasonable format for processing, it has to pass through a number of modules before it reaches the CPU. It would not have been possible to write all of these on my own and have them outputting data at such a high quality within the timespan of the project, so it was vital to use Altera blocks for pre-processing the video. The more interesting modules are discussed below.

Clocked Video In

Similar to the Video Out module, the Altera Clocked Video Input subsystem takes in composite video decoded by the DE2-115's TV decoder, dealing with clock domain crossing from the 27MHz TV-in signal to the rest of the system's 80MHz, and converting the data to the Avalon-ST VIP format the rest of the modules used, as described in Section 3.3.2.

Deinterlacer / Framebuffer

The data from the composite input is of resolution 640x240 and interlaced: the lines transmitted for each frame alternate, with every other line of data transmit-

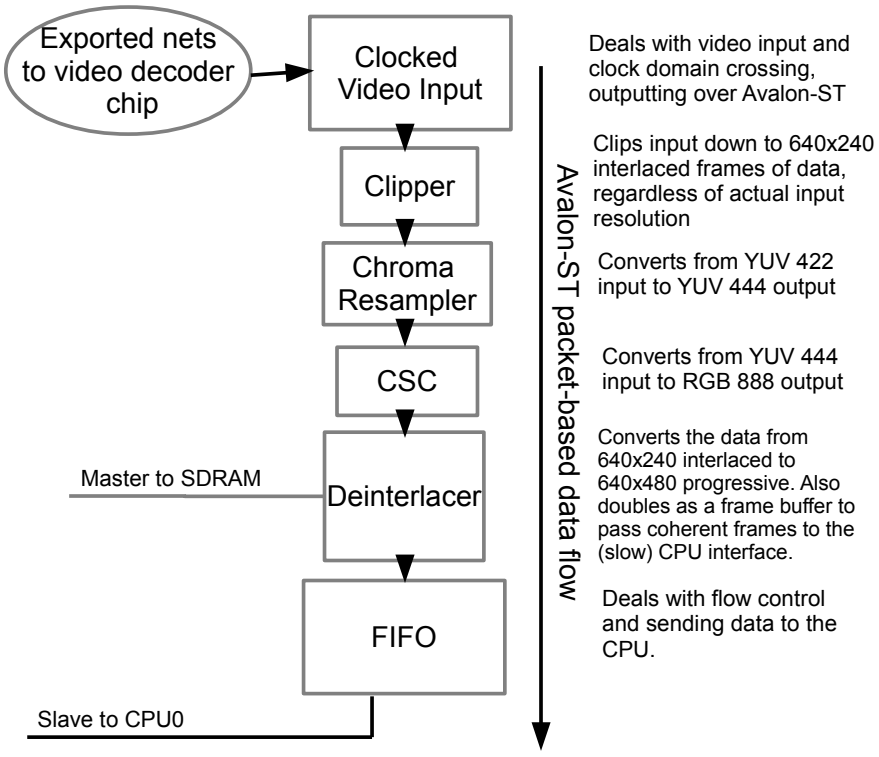

Figure 3.4: The structure of the initial video input system, with 8-bit FIFO programmed input to CPU

ted each frame. To achieve the correct format for output (640x480, progressive scan), we must fill in the extra data required. This implementation uses a hybrid approach: we double the lines when a part of the image has changed significantly, and otherwise use the line from the previous frame.

Since the above method requires data storage on DRAM anyway, the Altera module also allows the deinterlacer to act as a frame buffer for flow control: individual frames are stored in memory, which can then be output to the next module in the chain over the Avalon-ST streaming interface when the next module is ready for more data, discarding new frames while the current one hasn't been output.

3.5 Working Project – Initial

By this point, with the video in as described above, using a programmed IO FIFO, with one CPU performing MLAA on the input and outputting using the frame reader, I achieved a working project – I could load, process and output video data. This is shown in Figure 3.5. However, performance was far too low for the device to be usable. Simply loading in frames on the CPU via programmed IO took far too long: over a second for a single frame, and the MLAA step took over eight seconds for realistic input. This was not suitable for realtime interaction, but still, success criteria had technically been met by this point. What follows are improvements to this basic design to improve performance.

3.6 Optimisations to the Video In Subsystem

The following is a description of the techniques used to improve performance of video transfer from the Altera-ST devices into main memory to be processed by the CPU. Figure 3.8 shows the final video input subsystem.

3.6.1 Word Alignment

The Altera frame reader requires RGB data in a word aligned format: eight bits of R,G, and B, and eight blank bits per pixel. However, the Altera-ST streaming interface for video doesn't automatically support blank bits in its signal format. This meant that, for the initial implementation, I moved data around on the Altera-ST interface eight bits at a time. This was slow, and CPU intensive – every byte had to be processed by the CPU to place it in the correct position in memory, as well as add in the eight blank bits per pixel. Altera-ST supports

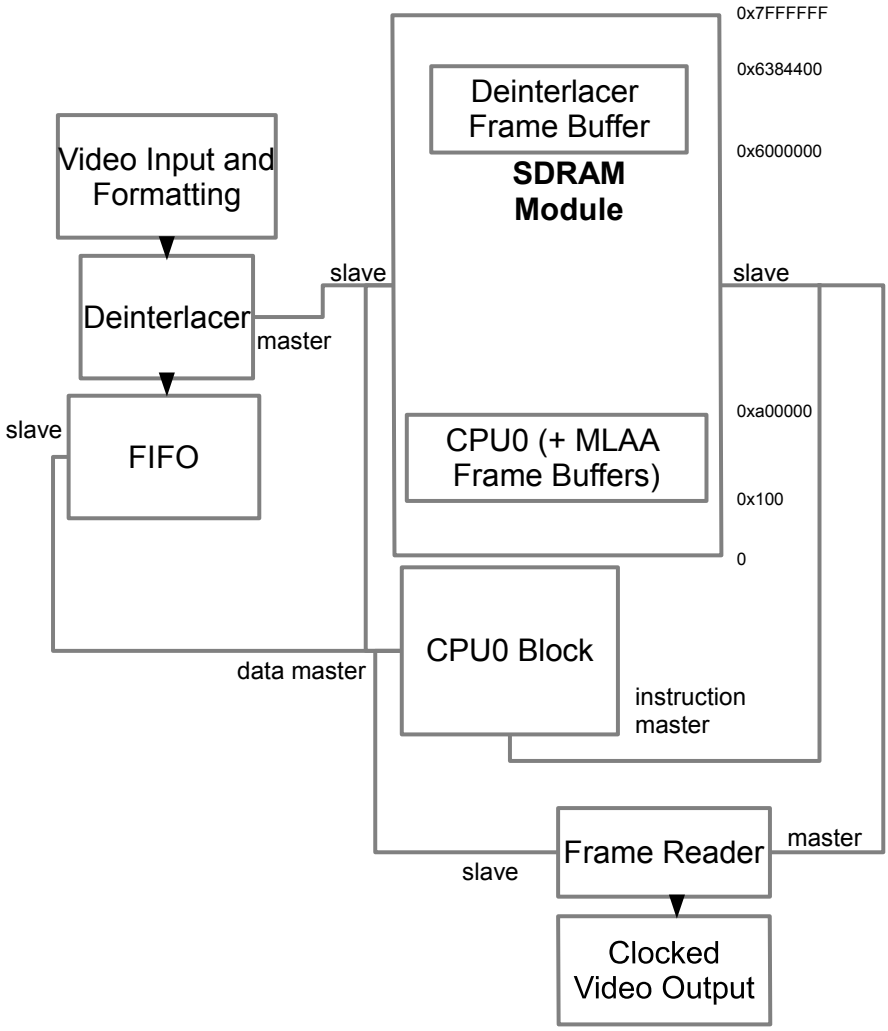

Figure 3.5: Structure of the first (slow) fully working system.

24-bit parallel formats, where all of the bits of a pixel are moved at once, however since this does not factorise into a 32-bit word this format was unsupported by any Altera programmed IO or DMA devices. Thus, I designed a Verilog module that took in Altera Avalon-ST 24-bit data, and added eight dummy wire bits (fixed at 0) to make the format 32-bit wide. Since the Avalon-ST video format is oppositely endian from the Nios-II soft-CPU, and it is trivial to change this in hardware (reassigning wires) compared to in software, I also used my Verilog hardware block to change the endianness of the video data. I imported this into QSys, as seen in Figure 3.6, setting relevant settings to work with Altera-ST, and this meant that I could simply add it in as a module to my System-on-Programmable Chip, as I would with Altera hardware.

The above processing of the data required register usage in the module, which delayed the signals from propagating by one clock cycle. This meant that, as Altera-ST is a packet based flow controlled protocol, ready and wait signals were delayed in the module by one cycle. This can cause problems through missing data, but the Altera-ST settings are configurable in QSys, and provision for dealing with such delay can be automatically synthesised provided the delay is specified in QSys.

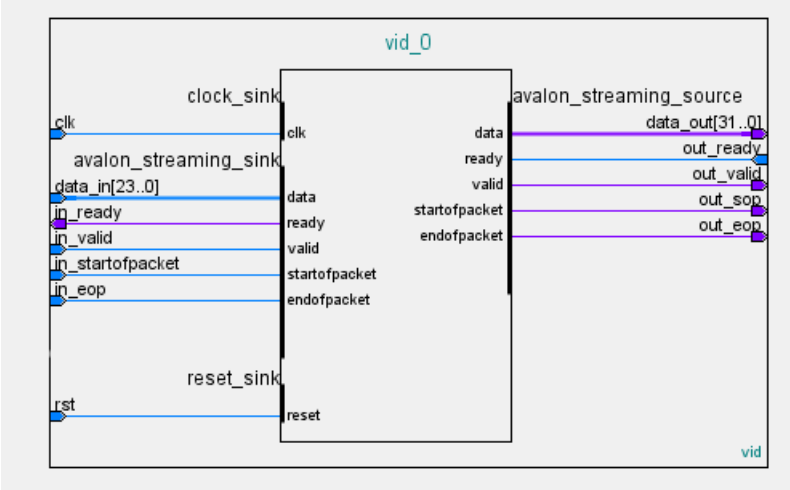

Figure 3.6: Qsys diagram of the data format converter, which changes the output from 24 bits to a 32 bit format suitable for DRAM.

3.6.2 Removal of Unnecessary Information

The Altera-ST video format is packet based: each frame can consist of multiple packets, of which two different kinds were transmitted from the deinterlacer:

video packets containing RGB video data, and control packets containing information such as resolution. Since the resolution of the video was fixed at 640x480 for this device, the latter were useless, and had to be discarded before MLAA processing began. Initially this was done in software, but I moved this to hardware by setting the output of my custom formatting module to "not ready" whenever control packets were being transmitted.

Each packet in the Altera-ST video format has a header in its first symbol, to indicate whether it is data or control. To discriminate between packet boundaries, start of packet and end of packet symbols are used. I used these to find control packets and remove them from the stream. However, since a video frame can be multiple packets long, and the first symbol of even the video packet is header data and not true video information, headers also had to be removed before placing the video frame into memory, otherwise video information would be skewed by a word by each packet. I also moved this into hardware, by delaying the ready and startofpacket symbols by one clock cycle when the start of a video packet was detected. This meant that all data going to the CPU via programmed IO was real, word aligned RGBX video data, and thus no formatting needed to be done on the CPU. A diagram of this can be seen in Figure 3.7.

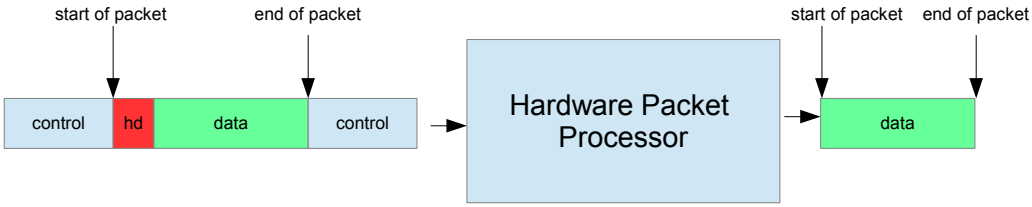

Figure 3.7: Diagram of the video packet processor, which removes control and header information from the video stream.

3.6.3 DMA Controller

The above optimisations of moving data formatting to hardware netted highly significant performance increases from over a second to around 290ms per frame (see Section 4.6). However, this alone, even without MLAA processing, was still too slow to actually make the device usable for playing video games. The problem here was that the upload was CPU-bound: even with word accesses the Nios II CPU was extremely slow at doing programmed IO and storing the results in

an SDRAM framebuffer. The logical choice, then, was to try to move this data movement to a memory controller which could do the upload to SDRAM by itself.

The initial implementation of the project needed processing on the CPU to word align and remove unneeded data from packets. However, the hardware described above removed this need, so a DMA controller was a good choice for moving data around faster. The standard Altera DMA only allows 64KB transfers per request, whereas a single 640x480x4 frame is 1.2MB. Having to make nineteen callbacks and interrupt receives to a DMA controller would likely itself be a huge bottleneck, so this was quickly discounted as being inadequate. However, a DMA controller suitable for transferring entire frames at once was found online [2], already integrated with QSys, and so by importing and using the DMA dispatcher and write master modules of that project I achieved loading in each video packet with a single request-interrupt cycle.

Using the DMA approach took frame loads down from 290ms to around 16ms (see Section 4.3.1), which was perfectly adequate for the purposes of this project.

3.7 Optimisations to the MLAA algorithm

While by this point the performance of loading frames into main memory to process them had been significantly optimised, the actual processing of the data with the MLAA algorithm was still incredibly slow, taking over nine seconds per frame. The following is a discussion of techniques used to improve the single core performance of the algorithm, before moving to discuss multi-core performance in the next section.

3.7.1 Word Alignment

Due to the fact that the bitmap format isn't word aligned, my original implementation used 8 bit `char` arrays to move through the image data. However, due to the RGBX32 format the Altera frame reader required, I was able to exploit 32 bit word-based optimisations on the Nios II implementation. For example, instead of checking for discontinuities per colour channel per pixel as I had to initially, I changed the check to use a bit mask such that only the three most significant bits of the colour in each word aligned pixel were exposed. I could then check if these words differed as a way to check for discontinuity: and given this operation is performed over a million times per frame of video (four times per pixel) such an optimisation was vital for good performance.

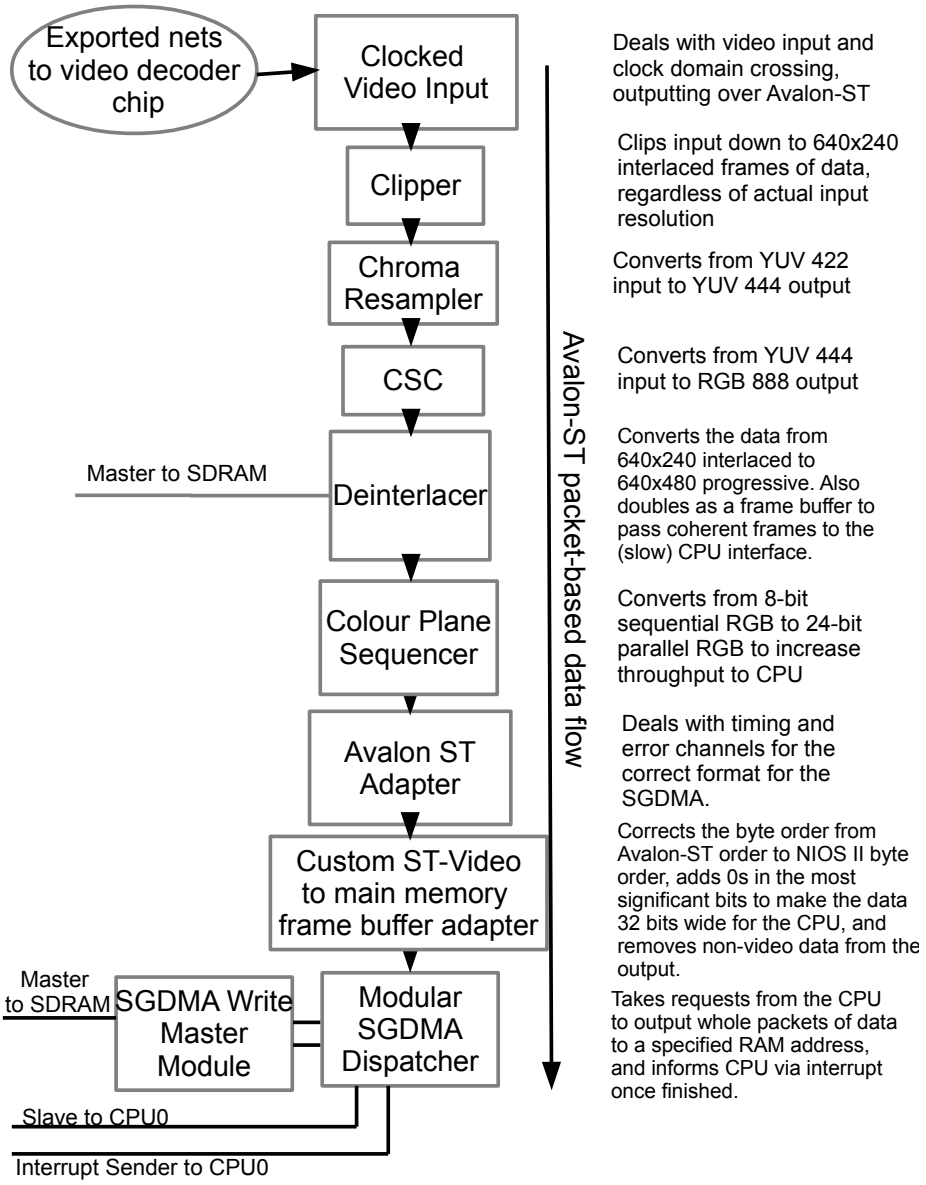

Figure 3.8: The structure of the final video input system, with DMA transfers to DRAM.

3.7.2 Fixed Point Conversion

My original implementation of the MLAA algorithm used floating point mathematics for working out the new colour of each pixel. Clearly this is inefficient compared to integer based calculations, especially on a CPU such as the Nios II, which features no floating point coprocessor, but correctness of the code was easier to justify. However, for performance I rewrote this part of the code using a 24.8 fixed point format: 24 bits of integer information followed by eight bits of fractional data. To justify the correctness of the new code I processed several images with both the floating point implementation and the fixed point one. Each was identical regardless of implementation.

3.7.3 Cache Awareness

In the initial implementation, separate scans of the frame data for horizontal and vertical discontinuities were made. This wasn't efficient for caching, as it didn't exploit temporal locality. I optimised the code by performing the same scan of the data in one big loop, resulting in a reduced IO cost: a technique called loop fusion. Similarly, I changed the dimensions of the data such that the innermost loops moved sequentially through memory; a technique called loop interchange, to exploit spatial locality better.

3.7.4 Loop Unrolling

The code for the MLAA algorithm I implemented featured many very tightly wound loops, particularly in the discontinuity checking stage. Thus, manual loop unrolling was implemented to reduce branching to improve the common case checks where we aren't near the edge of a given line, as well as decrease the number of memory accesses by reusing pixel values taken from main memory across pixel lines. By doing this manually, taking advantage of knowledge of the data, improvements over the compiler's own automatic loop unroller were achievable (see Section 4.6).

3.7.5 Custom Instructions

The advantage of working with an FPGA and soft processor over a more traditional fixed processor is that it can be customised for its particular function. My fastest software implementation of the function which generated the new colour of a pixel based on the fraction of each colour it contained, as seen in Figure 3.9,

took 27 instructions to calculate.[2] With a custom instruction, where custom logic is multiplexed with the Nios' stock Arithmetic Logic Unit, it was possible to do the same calculation in only three instructions, as seen in Figure 3.10 – a nine-fold improvement in speed per processed pixel. The custom instruction I made generated the colour given to the pixel by a particular side of the generated line. I used it twice and added the results together, resulting in the same output as given below.

This is a radical improvement in instruction count for such a frequently used code segment (potentially once per pixel = 300000 times per frame). The reason such an improvement was possible was because such an operation is very far removed from the instructions available on a RISC processor: we need to multiply each eight bit colour segment of each pixel separately to achieve the correct effect, as well as shift each 8-bit colour segment to its original place after processing. By doing this in hardware much of the work can be done in a single cycle by multiplying using larger values than words (to ensure the colour channels don't merge after multiplication), accessing colour bits using wires instead of bitmasks, and moving bits back to their original place by assigning wires instead of bit shifting.

The Nios II supports both single cycle and multi-cycle instructions, and my original naïve implementation of the custom instruction described above used a separate multiplier for each colour channel, and bit masks for accessing each colour. This initially took four cycles to avoid breaking timing constraint, but I later optimised the instruction to be executed in a single cycle by using wires to mask data, and a single 48-bit multiply. To verify the correct result was being calculated by these custom instructions, I wrote a testbench that used a purely software implementation and the hardware accelerated version of the calculation, and compared the results. To ensure future alterations to the hardware didn't break this correctness, this testbench runs at the start of the program.

3.8 Multi-core Implementation

With no more obvious choices for single core optimisation, I turned my attention to exploiting the highly parallel nature of the algorithm (as discussed in Section 2.2) by processing on multiple cores.

[2]This figure was calculated from looking at the assembler objdump the compiler created with optimisations enabled.

```
((unsigned int*) image->image_data)[x1 + y1 * 640] =
    (((pixel2 & 255) * alpha + (pixel1 & 255) *
    ((1 << 8) - alpha)) >> 8) + ((((pixel2 & (255 << 8))
    * alpha + (pixel1 & (255 << 8)) * ((1 << 8) - alpha))
    >> 8) & (255 << 8)) + (((((pixel2 & (255 << 16)))
    * alpha + (pixel1 & (255 << 16)) * ((1 << 8) -
    alpha)) >> 8) & (255 << 16));
```

Figure 3.9: The fastest software implementation I achieved of generating the new colour of a pixel in my algorithm. Note its unwieldy nature and large operation count.

```
340: 6adbc032  custom 0,r13,r13,r11
344: 62d9c032  custom 0,r12,r12,r11
348: 6b19883a  add r12,r13,r12
```

Figure 3.10: The Nios II assembly generated using custom instructions equivalent to the expression in Figure 3.9.

3.8.1 Mutual Exclusion

To synchronise multiple processors, Altera provide a mechanism for mutual exclusion by providing a hardware mutex: the idea is that each Nios II soft-CPU has its own ID, and can use that ID to gain a lock on the mutex. I based my usage of the Altera hardware mutex on an Altera tutorial [3]. However, since what I was actually wanting to provide was a software mutual exclusion system for access to memory, this was not enough on its own: my software mutex system used this hardware mutex, together with a clearing of cache each side of the lock and unlock commands. To avoid the compiler reordering these instructions via an optimisation, I used the memory barrier instruction `asm volatile("" ::: "memory");` to keep memory instructions on the correct side of each lock.

3.8.2 Multiple Processors

While the original CPU used up until this point had to deal with loading in data from the DMA engine, outputting it again over the frame reader, measuring execution speed and allocating memory to store each frame in memory, the other CPUs would only have to deal with processing their fraction of each frame. Thus the devices connected to each CPU differed, as did the amount of instruction and data cache devoted to each one: 8KB and 32KB instruction and data cache for

CPU0, and 2KB and 16KB for CPUs 1-11. The structure of the hardware for each CPU is shown in Figure 3.11 for the primary CPU and Figure 3.12 for the secondary CPUs. Each processor was connected as a master to main memory: the Altera Avalon-MM memory mapped network on a chip deals with arbitration automatically. Each CPU has its own memory assigned for its stack, program data and heap in the address space, but can access other CPUs' memory: this is used so that every CPU can access the buffers allocated by the main CPU.

3.8.3 Inter-core Communication

To communicate information on frame address, state of the system, mutex and shared discontinuity buffers, I used a mailbox design: each CPU can read and update the struct described in Figure 3.13, subject to mutual exclusion on writes, which is stored at the fixed physical address 0: to ensure it couldn't be written over, I placed the mailbox in physical memory at an address below any processor's physical address space.

3.8.4 Multi-core Programming on Nios

Multi-processing is supported on the Nios II by using separate binaries for each processor, each uploaded over a separate JTAG UART interface. This is quite unlike any conventional desktop multiprocessing system, where code for each processor is included in a single binary, so required a more explicit allocation of work between processors at compile time.

Each processor in my implementation processes $\frac{1}{n}$ of the horizontal and vertical lines of the image of each stage, where n is the number of processors. This allows us to split up the work at each stage fairly evenly, as well as making mutual exclusion fairly simple: we wait after each processor has finished each stage of the algorithm before moving on to the next. The same code for morphological anti-aliasing was used on each processor: to allow each processor to process particular parts of a frame, and to allow the main CPU to control the processing, the C preprocessor was used together with the CPU ID defined in each processor's project to specialise the compilation for a given CPU. This helped to provide uniformity across CPU implementations, thus avoiding bugs in the code through inconsistency.

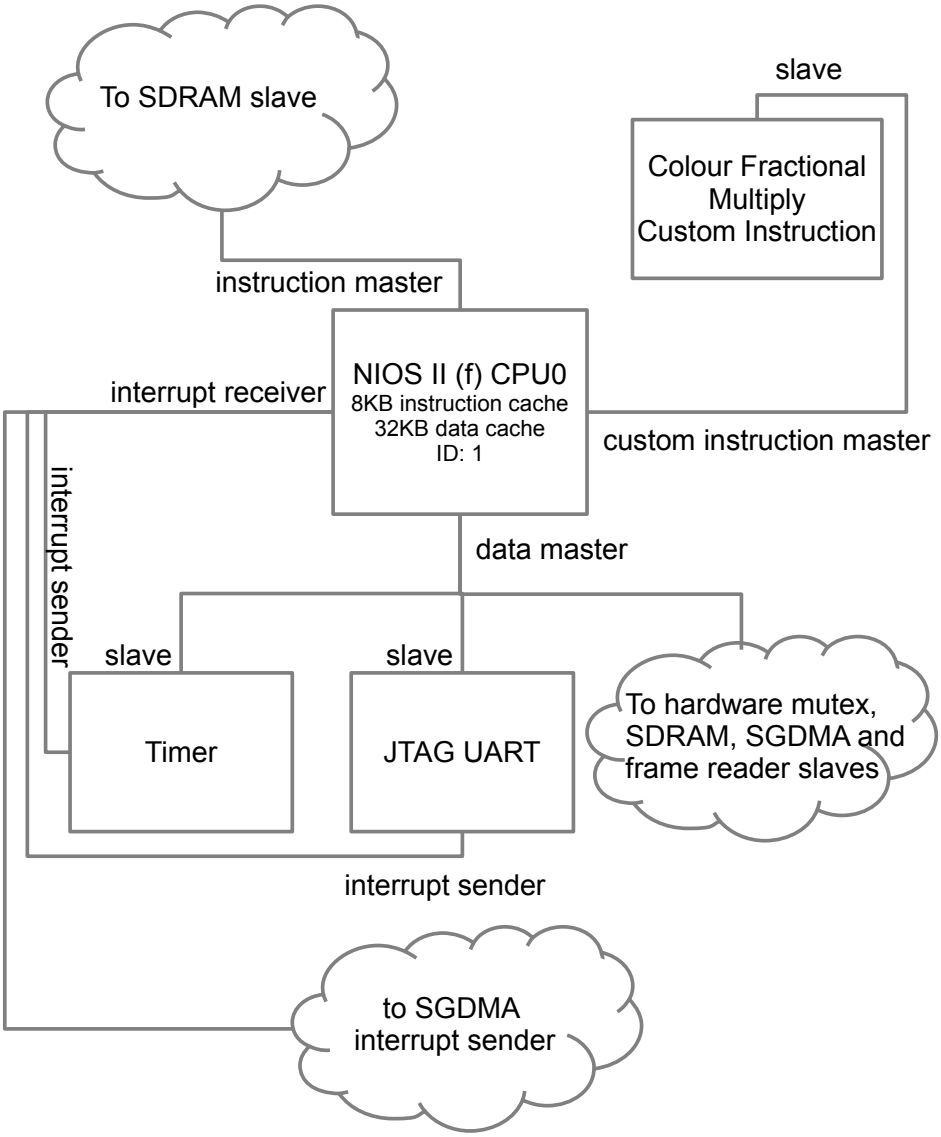

Figure 3.11: Structure of the components attached to the main CPU.

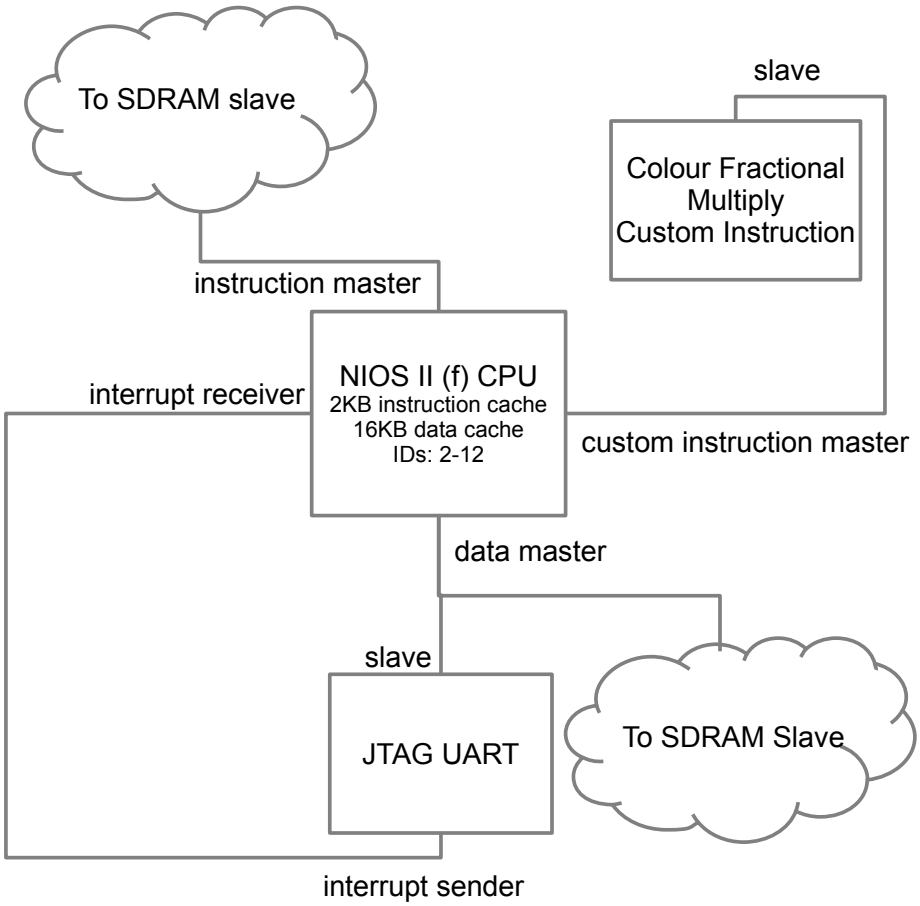

Figure 3.12: Structure of the components attached to the secondary CPUs.

```
typedef struct system_info {
      unsigned int* current_framebuffer;
      unsigned int started;
      unsigned int volatile procs_waiting_to_find;
      unsigned int volatile procs_waiting_to_proc_horiz;
      unsigned int volatile procs_waiting_to_proc_vert;
      alt_mutex_dev *mutex;
      _Bool (* vertical_discontinuities)[639];
      _Bool (* horizontal_discontinuities)[640];

} system_info_t;
```

Figure 3.13: Structure of the mailbox to communicate between processors, stored at address 0.

3.9 Final Device

The final device is shown in Figure 3.14, complete with all optimisations described above, twelve processors, custom instructions, DMA engine for loading frames, and multi-core synchronisation. The Figure also shows the locations in physical memory of the data stored by each part of the system.

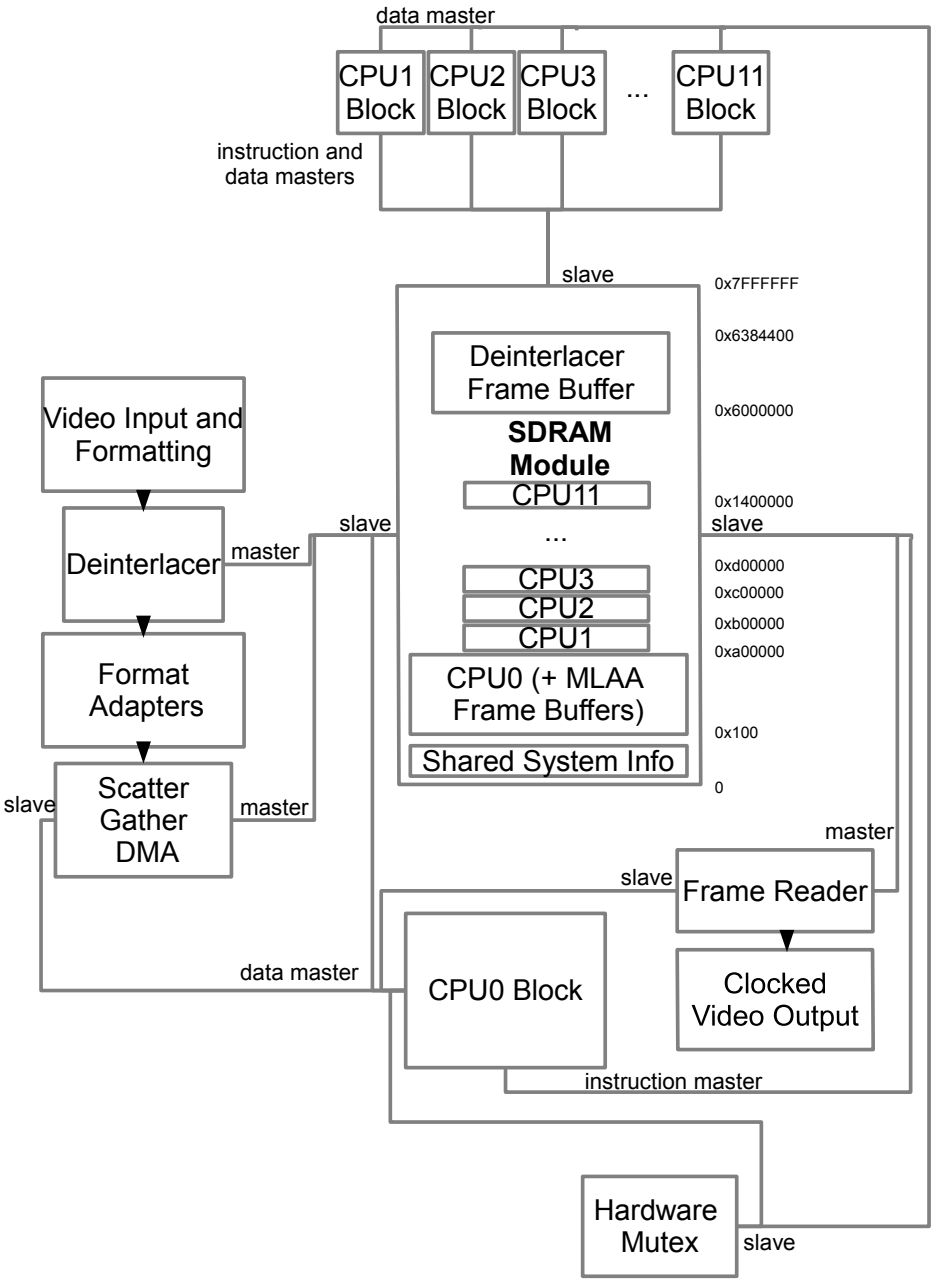

Figure 3.14: Structure of the final system.

Chapter 4

Evaluation

4.1 Output of Morphological Anti-Aliaser

Figure 4.1 shows an example of sample output[1] from my bitmap implementation of MLAA, with Figures 4.2 and 4.3 comparing the image to alternative anti-aliasing techniques (super-sampling and a blur filter respectively).

As this image features very well defined discontinuities, it is a fairly ideal case for the algorithm. However, arbitrary aliased images also show strong improvement in image quality; the extent to which will not be discussed here as it is by nature subjective, but is discussed in detail in the original MLAA paper [11].

One thing we may wish to consider are the benefits and disadvantages of applying this filter to the video stream rather than on the hardware rendering the device. Performing the technique on the original device uses power of the system that could be used for other purposes, and isn't possible when we don't have access to the hardware (as is common with games consoles). However, the algorithm often has relatively poor performance on text and on-screen display information, and as a result the process is generally applied before such information is overlaid in a rendering pipeline [11]. This is not possible when the only available input is the whole video frame, and thus some detail is lost in such text as a result.

The output video stream from a device is sometimes not at the same resolution as the frame it originally rendered; upscaled 3D is common to improve performance. In this case the performance of the algorithm is limited as the pixel based discontinuities we are trying to clean up are actually larger than a single pixel. This problem is easy to work around on the device rendering the graphics, as the filter can be applied before upscaling, but harder when working with just a video stream.

[1]The source of the image used is a screenshot from *Yet Another RayTracer for .NET* [9]

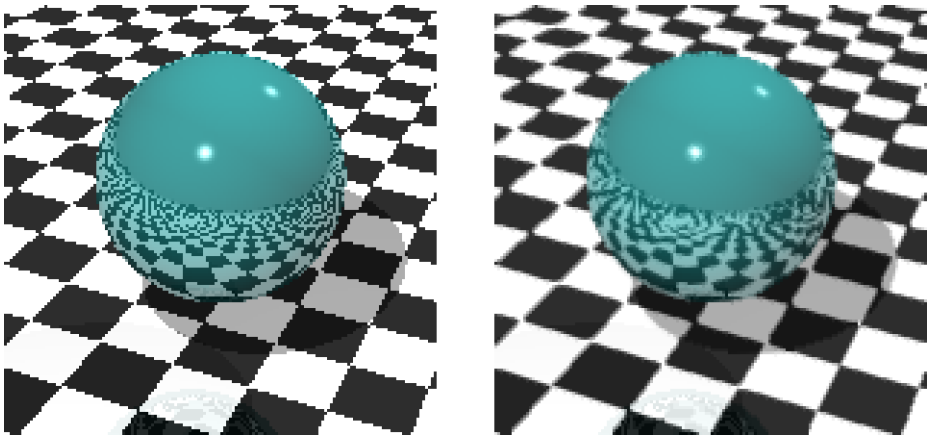

Figure 4.1: Original image, and image passed through my morphological anti-aliaser.

Still, it is extremely challenging to do better without access to the rendering hardware itself. A potential improvement would be to do detection of on-screen display information, to avoid blurring it, but this would be a significant undertaking at little visual gain, and potentially show limited parallelism compared with the MLAA used in my implementation.

4.2 Output of device

Figures 4.4 and 4.5 show screenshots from a demonstration video I made to show off the device running on the video output of a Wii console. The performance under different scenarios varies from around 8-15 frames per second depending on number of lines fitted: this particular scene could be processed at around 12 frames per second. As video games typically render between 25-30 unique frames per second, some temporal information is lost as a result of processing being too slow: however, it should be expected that the same device running on a faster FPGA could process and output every unique input frame fairly easily.

Still, improvements in image quality are limited in this case by the input signal. The only possible video input on the DE2-115 (and most FPGA development boards) is composite in, which is both interlaced and limited to standard definition input. More severely, due to the physical properties of composite connections they also exhibit dot crawl and colour bleeding as a result of high frequency luminance information crossing into the colour component of the signal[2]. A more

[2]See [5] for a visual description of the loss of quality as a result of composite video

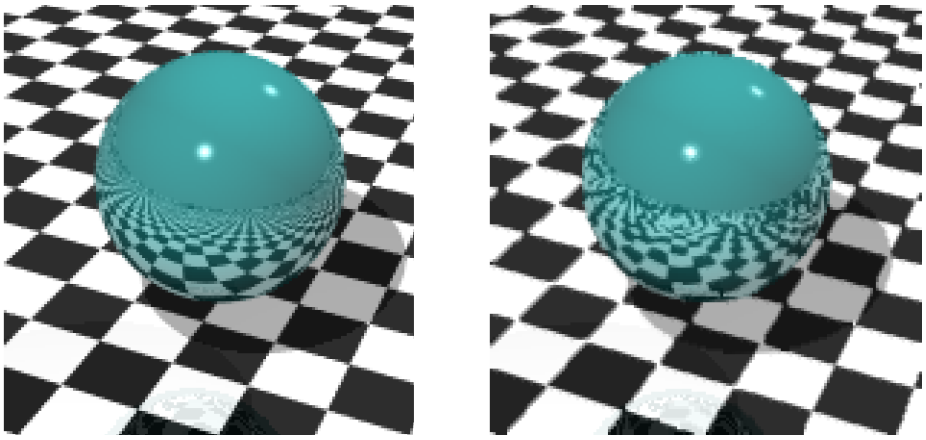

Figure 4.2: Raytraced image supersampled using Monte Carlo sampling (L) compared to the morphologically anti-aliased image (R) as above. The result is very similar.

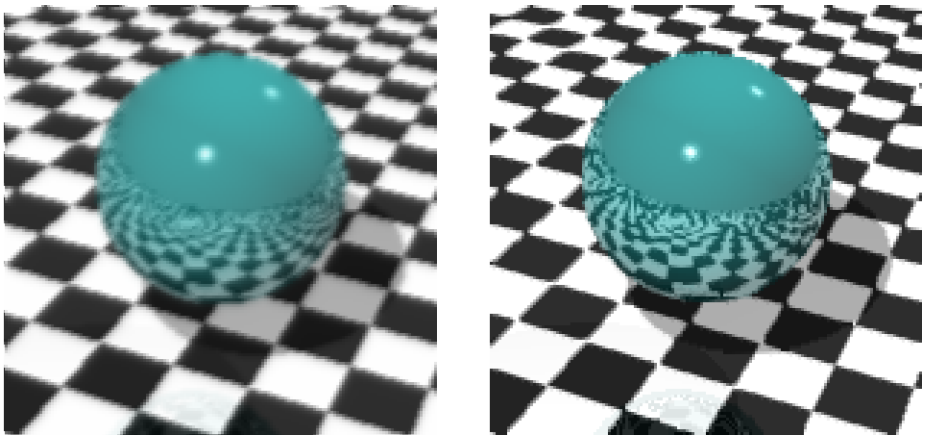

Figure 4.3: The original image Gaussian blurred (L), versus the same image with MLAA applied(R). The former results in a significant loss of detail compared with the latter.

Figure 4.4: A screenshot of the device filtering the composite output of a Nintendo Wii console and outputting it over VGA.

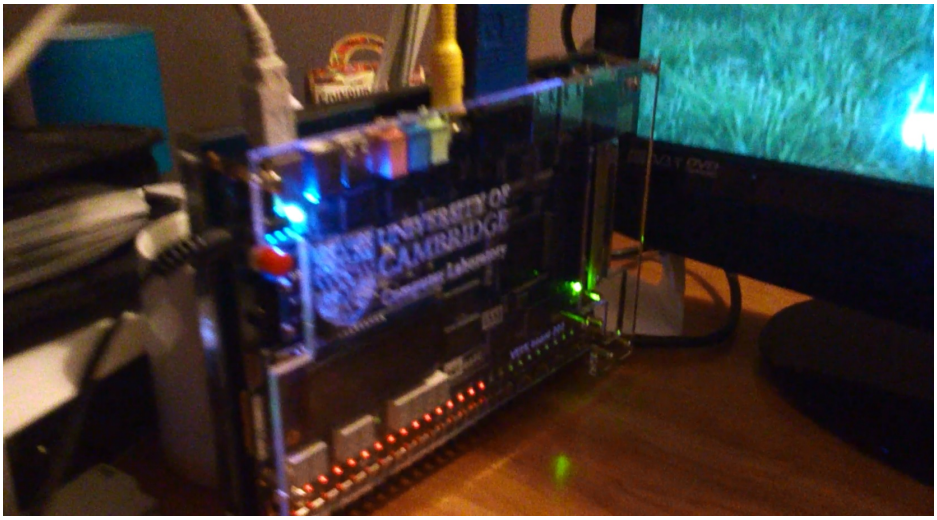

Figure 4.5: The Altera DE2-115 doing the processing shown in Figure 4.4.

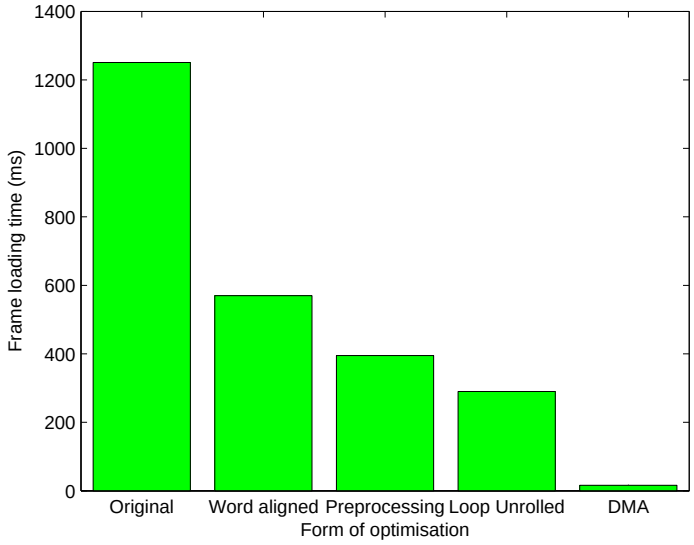

Figure 4.6: The time per frame with various optimisations successively enabled for the loading of video frames, from least to most optimal. Preprocessing entails removal of headers and control packets in hardware, and DMA is the final implementation.

modern format such as HDMI would allow input of the data into the system at higher resolution and without artefacting, but was unavailable to me on an FPGA development board for the purposes of this project.

4.3 Performance

4.3.1 Frame Loading

Figure 4.6 shows the average frame load times to get frames into DRAM at each stage of implementation mentioned in Section 3.4. What's notable here is that we go from just over 1200ms to 16ms between the slowest and fastest implementation – performance here improved by almost two orders of magnitude as a result of the optimisations I performed.

Figure 4.7: The output used to test the speed of the MLAA implementation.

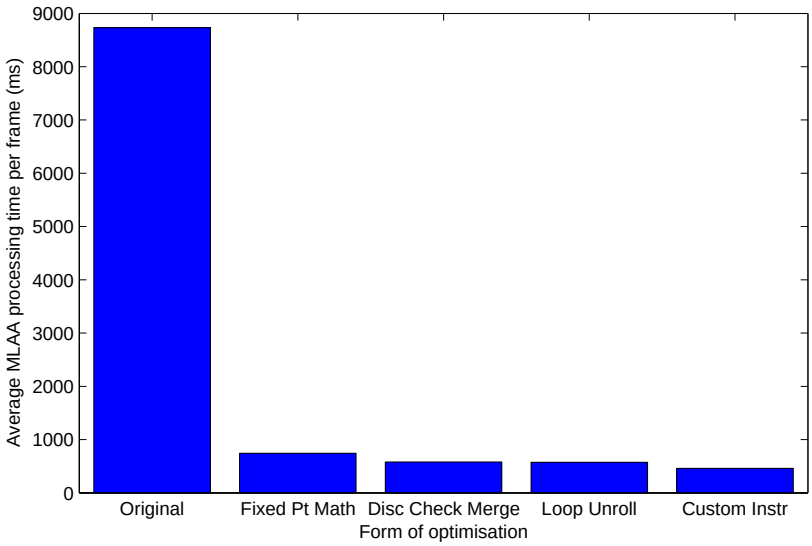

Figure 4.8: The MLAA processing time per frame with various optimisations added.

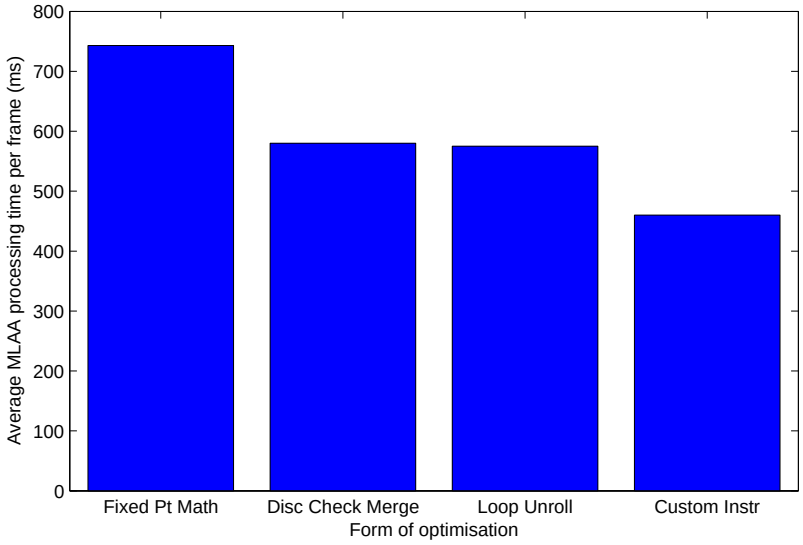

Figure 4.9: The MLAA processing time per frame with various optimisations enabled, with reference to the fixed point implementation

4.3.2 Morphological Anti-Aliasing

Single-Core Performance

Figure 4.8 shows the difference in MLAA processing time per frame on the screen shown in Figure 4.7 with respect to the optimisations from Section 3.7. Figure 4.9 shows the same graph without the original floating point version, as the difference between that and the next fastest implementation is so large it is difficult to see any other changes on the first graph. From slowest to fastest single processor implementation, we have a difference in speed of almost 20 times. The float-to-int conversion gave the largest improvement, which is to be expected as the Nios lacks a floating point unit, followed by the merging of discontinuity checks by word alignment and custom instructions, as both massively reduce the amount of CPU instructions performed. Loop unrolling the discontinuity check gave only a small improvement (around 5ms per frame), as similar optimisations performed by the GCC based compiler were likely already quite good.

Multi-Core Performance

Figure 4.10 shows the MLAA processing time per frame of the screen in Figure 4.7 with n cores. We see that by going from one to twelve processors we go from

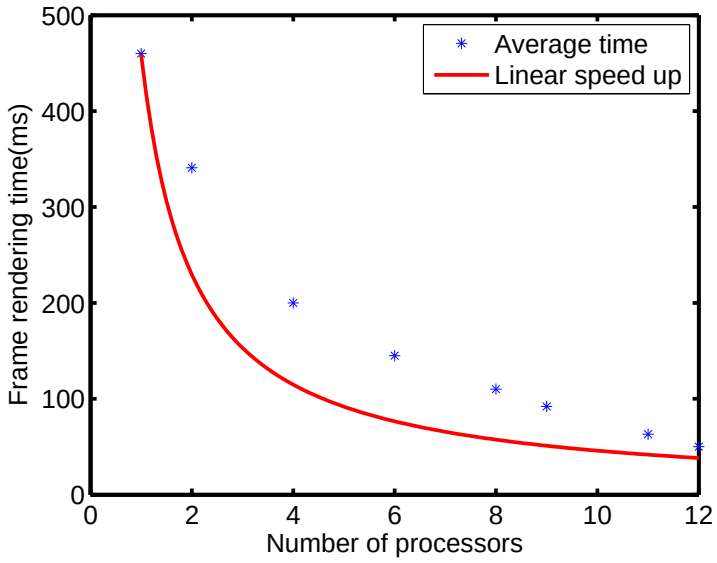

Figure 4.10: The MLAA processing time per frame with n cores, on the screen from Figure 4.7, with ideal linear speedup line.

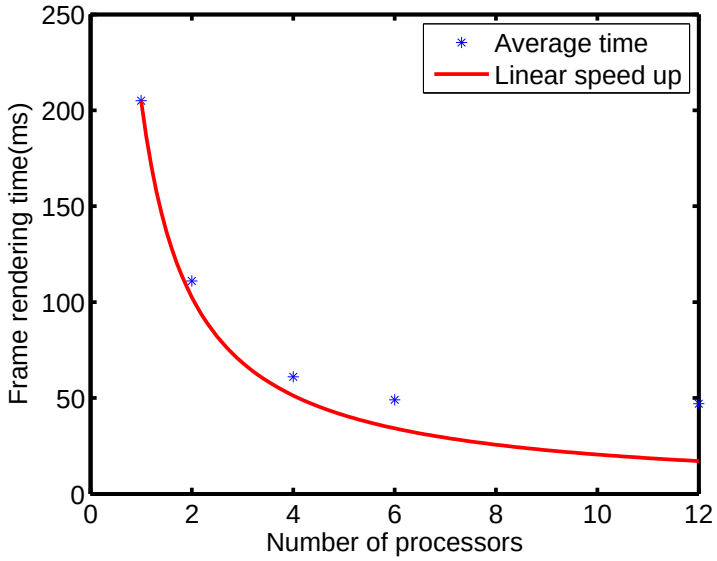

Figure 4.11: The MLAA processing time per frame with n cores, on a blank screen, with ideal linear speedup line.

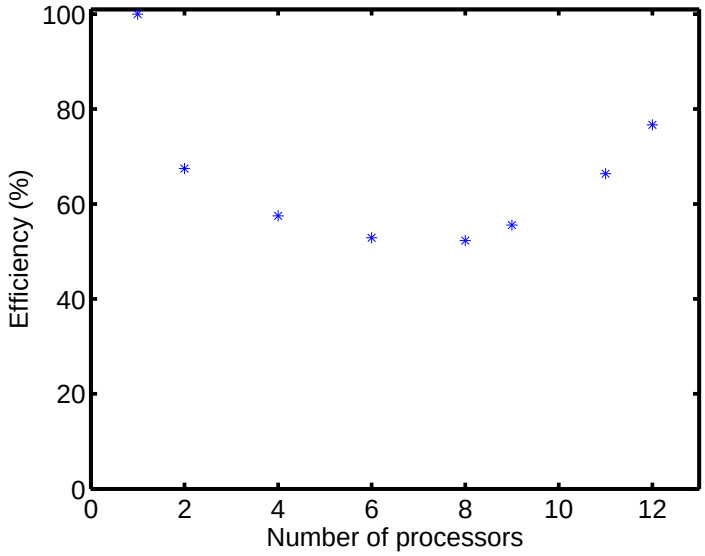

Figure 4.12: Strong parallel scaling efficiency of MLAA with n cores (processing Figure 4.7).

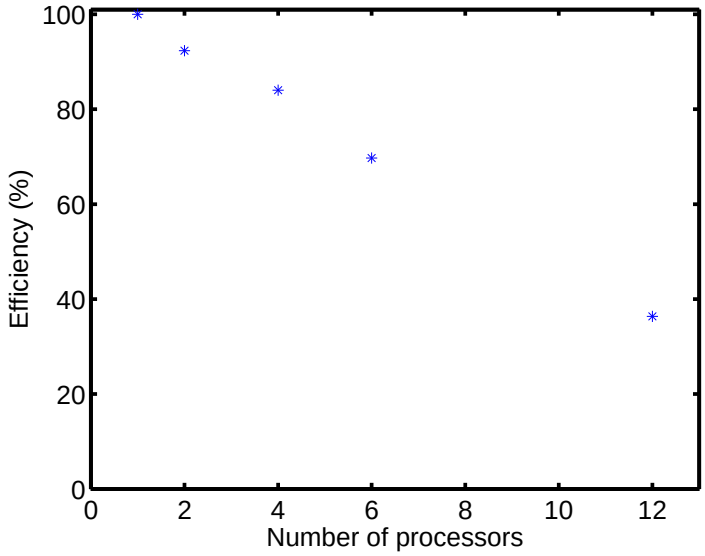

Figure 4.13: Strong parallel scaling efficiency of MLAA with n cores (processing blank screen).

spending 460ms per frame to 50ms per frame – a 9.2x improvement. A perfect linear speed up (and perfectly parallel computation) would give us twelve times improvement.

However, we see that even with the high theoretical parallelism of the algorithm, we lack perfect linear speedup due to the limited memory bandwidth shared between every processor. Figure 4.11 shows frame processing times for blank screens – where no lines need to be fitted. We get good speed up by adding more cores at first (where we are CPU bound), but this quickly tails off at around six cores to 45-50ms. This is because we become data bound – we are limited by the speed of the RAM rather than parallelism of the algorithm.

Indeed, Figure 4.13 shows the strong parallel scaling efficiency (calculated as $\frac{t_1}{N \times t_N} \times 100\%$ where t_i is time taken on i cores) of processing the blank screen. We see that where we are memory speed limited, efficiency drops off rapidly.

What of the case of the test screen in Figure 4.7? The efficiency per processor here, as seen in Figure 4.12, doesn't drop off as quickly, reducing only to 77% with twelve processors. This is because we are actually processing discontinuous lines in this case, thus performing more computation, and we are thus more computation bound. Still, given the closeness in performance for twelve cores between this and the blank case (47ms vs 50ms) it is likely that adding more cores than twelve would give little gain, and efficiency would thus drop quickly. We should consider the shape of the graph. Counterintuitively, for 1–12 cores efficiency goes down up to six cores before shooting up again. This is an unexpected result – with sharing equal work between processors scaling efficiency should be monotonically decreasing. We could consider this to be caused by uneven computation between cores: each core is given $\frac{1}{n}$ of the pixels on screen, rather than $\frac{1}{n}$ of the pixels that have lines to process; the blank screen case in Figure 4.13, where work is equal per core, showing monotonic decrease compared to the more uneven natural image case in Figure 4.12 would support this. However, an identical increase in efficiency from 6–12 cores was shown using a random noise input, where the amount of lines processed per core was roughly the same, and total work done per frame was verified to be constant regardless of the number of cores used. It is likely, then, that the increase in performance is simply an artefact of bus contention over Altera Avalon.

4.3.3 Latency

I mentioned in Section 2.1 that low latency was important for real world use of this device. Given a television would have to perform many of the in-hardware time consuming tasks such as deinterlacing if this device didn't perform them, we

can consider time through CPU to be a good proxy of the total delay caused by the processing. For the test in Figure 4.7 with twelve CPUs, this totalled around 66ms – displays and consoles together typically add around 166ms latency before this [10], so the additional latency here isn't excessively large. Indeed, sound is not delayed by this processing, and yet doesn't appear to be out of sync with the displayed video.

4.4 Evaluation of Development Process

The iterative development model I chose during the planning stage was vital in the success of the project: the only way I was able to verify each stage of development was by adding features steadily and testing each new addition. The early development of a working (but slow) version of the device meant I could take performance measurements very early on in the project, making it clear where next in the system to target performance improvements at each stage.

The use of Altera components helped enormously; having to design my own hardware to do video decoding would have absorbed much of the time of the project, meaning much less time would have been available for performance extensions, and thus that the end result would likely be unusable for any realistic use case. By comparison, the 8-15fps achieved by the end of the project still allows low enough latency for games while improving image quality.

The project didn't always go to plan with the original proposal document: I had initially planned to implement a multi-core PC version of the MLAA algorithm. However, the research into the style of parallelism available with Altera's systems gave me signs early on that this would have been useless at giving design direction for the final FPGA implementation, and thus I was able to save on some work that would later turn out to be useless.

Chapter 5

Conclusion

My project was, overall, highly successful. Not only did I achieve the processing of video set out in the specification, I managed to achieve an over 140X increase in speed over my initial implementation as a result of code optimisations, custom hardware and exploiting parallelism. The final device meets the success criteria set out in the project proposal; it accepts video input over composite, and displays output over VGA filtered by the MLAA algorithm. Further, I extended the project by increasing performance of my device to the point of achieving low enough latency and high enough frame rate to be usable in real life contexts. The project also displays video at the full spatial resolution of the original signal, and uses custom hardware to improve performance, as described in my original extensions plan in the proposal.

5.1 Limitations

The project was limited by the performance of the FPGA development board; the DRAM bandwidth was, in the end, a key limiter of processing speed, achievable clock speeds were relatively low (80MHz), and the composite input limited the ability to output high quality video. It is likely that a higher power FPGA development board with high quality video input would solve all of these issues.

5.2 Future Work

This project was limited by the hardware available. It may be interesting for future implementers of similar projects to try other methods to improve performance and quality:

5.2.1 HDMI Input

The quality of composite input, due to its low resolution, interlaced nature and colour bleeding, is very low compared to HDMI. FPGA development boards exist with such input connections, and would allow high quality lossless HD video to be processed by the MLAA algorithm.

5.2.2 Super FPGAs

It is now common for FPGAs to come with dedicated, high performance processors on-silicon – so called Super FPGAs. These are now available at fairly low cost – the Altera DE1-SoC development board for example contains two ARM Cortex A9 processors running at 800MHz, along with high speed DDR3 memory. Utilising these would limit parallelism and the ability to use custom hardware instructions, but it is plausible that the high clock speed and instructions per cycle compared to Nios II would allow for higher performance.

5.2.3 Full Hardware Implementation

Using software on CPUs helped reduce the development time of this device significantly, but it is possible performance and power consumption would be improved with a full hardware implementation. Bluespec Verilog would be a good candidate high level language to manage the inherent complexity of designing such a system.

Bibliography

[1] DE2-115 TV to VGA sample project. http://www.alteraforum.com/forum/showthread.php?t=31630.

[2] Modular SGDMA. http://www.alterawiki.com/wiki/Modular_SGDMA.

[3] Altera. Creating multiprocessor Nios II systems tutorial. http://www.altera.co.uk/literature/tt/tt_nios2_multiprocessor_tutorial.pdf.

[4] Intel Corporation. Morphological antialiasing: Sample project. http://software.intel.com/en-us/vcsource/samples/morphological-antialiasing-mlaa.

[5] Chris Covell. A little bit about NTSC colour encoding. http://www.chrismcovell.com/gotRGB/screenshots.html.

[6] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[7] Jorge Jimenez, Diego Gutierrez, Jason Yang, Alexander Reshetov, Pete Demoreuille, Tobias Berghoff, Cedric Perthuis, Henry Yu, Morgan McGuire, Timothy Lottes, Hugh Malan, Emil Persson, Dmitry Andreev, and Tiago Sousa. Filtering approaches for real-time anti-aliasing. In *ACM SIGGRAPH Courses*, 2011. http://iryoku.com/aacourse/downloads/Filtering-Approaches-for-Real-Time-Anti-Aliasing.pdf.

[8] David Kirkby. BMP format. http://atlc.sourceforge.net/bmp.html.

[9] Herre Kuijpers. Yet another RayTracer for .NET, 2007. http://www.codeproject.com/Articles/15935/Yet-Another-RayTracer-for-NET.

[10] Richard Leadbetter. Console gaming: The lag factor, 2009. http://www.eurogamer.net/articles/digitalfoundry-lag-factor-article?page=3.

[11] Alexander Reshetov. Morphological antialiasing. In *Proceedings of the 2009 ACM Symposium on High Performance Graphics*, 2009. http://visual-computing.intel-research.net/publications/papers/2009/mlaa/mlaa.pdf.

[12] Peter Thoman. PtBi. http://ptbi.metaclassofnil.com/.

Appendix A

Project Proposal